Object-Oriented VRML For Multi-user Environments[*]

Sungwoo Park, Taisook Han[†]

Department of Computer Science, KAIST[‡]

Abstract

We suggest a new 3D scene description language which is a derivative of VRML (Virtual Reality Modeling Language). The language enhances the object-oriented feature of VRML, especially with respect to programming the behavior of nodes. The language no longer provides the Script node and the route. Instead, event handlers enable a node to take general actions. The event handler is included in the definition of node prototypes. The language has a feature that supports the multi-user environment. One of the design criteria is its suitability for the easy implementation of multi-user virtual environment systems. For this purpose, we propose a new structure of messages distributed among clients in a multi-user system.

CR Categories and Subject Descriptors: I.3.6 [Computer Graphics]: Methodology and Techniques - Languages; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - Virtual Reality.

Additional Keywords: object orientation, multi-user environment, multicast mechanism.

1 INTRODUCTION

The Virtual Reality Modeling Language, VRML [2, 3], is a file format for describing interactive three-dimensional scenes. It has developed from a simple 3D scene description language to a more sophisticated 3D modeling language capable of expressing behavior and supporting interactions. VRML 2.0 provides many extensions and enhancements to its predecessor, VRML 1.0, and has become the standard 3D modeling language.

With its evolution to VRML 2.0, VRML has included some new object-oriented graphic constructs. Object-oriented features of VRML 2.0[1] are such as prototype extension mechanism, event passing among nodes, and behavior associated with nodes. However, some of the language features violate the general object-oriented design principle. For example, adding behavior to a node is accomplished by exploiting Script nodes and routes that are not a part of the node. A Script node is an independent object in VRML which can be connected to other nodes only through routes. Thus, the behavior of a node is controlled not by its own elements but by other independent nodes, namely, Script nodes in conjunction with routes. If the positions of Script nodes and routes in a VRML file are not chosen carefully, one may find difficulty in understanding the behavior of a node to which Script nodes and routes are related. It is due to insufficiency of object-oriented features in VRML. Adding further object-oriented features to VRML will make it easier to design and create a complex virtual world. Also, one will be able to understand VRML files more easily.

VRML does not have many language constructs to support multi-user environments. One of its design criteria, 'multi-user potential', implies that it has not yet incorporated language constructs designed specifically for multi-user environments. Currently, a variety of multi-user virtual environment systems based on VRML are available. However, there exists no general agreement on the basic structures, such as message transmission protocol for a server and its clients or contents of a message, on which those systems should be established. It is primarily attributed to deficiency of language constructs for supporting multi-user environments in VRML.

The main goal of this paper is to introduce a new object-oriented 3D scene description language supporting the easy implementation of multi-user virtual environment systems. We call this language OO-VRML (Object-Oriented VRML), which is a variant of VRML. In Section 2, we explore the general concept of object orientation. In Section 3, the object-oriented features of OO-VRML are explained as we present the way of defining a new node type. The comparison of VRML and OO-VRML clarifies the main differences of two languages. In Section 4, we show the characteristics of OO-VRML which support multi-user environments. In Section 5, we present how to represent a new node with a complete example. In Section 6, we show that OO-VRML is an improvement on VRML. Finally, we summarize this paper and present future directions.

2 OBJECT ORIENTATION

Before presenting the proposed syntax of OO-VRML and its semantics, it would be helpful to examine the general notion of object orientation. Object orientation [7, 8] is both a language feature and a design methodology. In both cases, an object is composed of a set of operations on some encapsulated data of its own. Regardless of its size and type, all interactions with an object occur only through simple operations called 'event sends'. An event send applied to an object invokes the associated operation of the object and causes the object to take an action (or behavior).

The node of VRML is a language construct corresponding to the object of object-oriented systems. Yet, every action of a VRML node is accomplished simply by assigning the argument in an incoming event to the associated field of the node. In other words, a node is controlled practically by other Script nodes or event-generating nodes that are connected to the node via routes, which violates the basic principle of the object-oriented design. Hence, in order to improve object-oriented features of VRML, it is important that all actions of a node be defined inside the node and that the description of the actions be invisible from the outside.

A typical object-oriented language provides such features as subtyping, inheritance, and encapsulation. In VRML, subtyping means that if some node *n1* has all of the functionality of another node *n2*, we may use *n1* in any context expecting *n2*. Inheritance is the ability to use the definitions of simpler nodes in the definition of a more complex node. Encapsulation means that access to some portion of a node's data is restricted only to that node.

All these features are incorporated into VRML, either partially or completely. For example, the first node found in a prototype definition is used to determine the actual type of nodes from this prototype when used in a VRML file, which shows the complete subtyping feature of VRML. However, encapsulation of VRML is provided in a very restrictive form because fields of any node except Script nodes can be either simply read or simply written to, i.e. operations to be performed on those fields are limited. The first case is found in the use of `IS` syntax in the prototype definition. The second case occurs when an event is generated in a node and the event is delivered to another node via a route.

---

[*] This work is supported by the ETRI, Electronics and Telecommunications Research Institute, under grant 96-09-960520. This work is supported in part by Korea Science and Engineering Foundation (KOSEF) through Center for Artificial Intelligence Research (CAIR), the Engineering Research Center (ERC) of Excellence Program.

[†] e-mail address : `{gladius,han}@compiler.kaist.ac.kr`

[‡] Korea Advanced Institute of Science and Technology, 373-1 Gusung Yusung Taejon South Korea 305-701

[1] VRML indicates VRML 2.0 hereafter.

In most cases, general operations, such as arithmetic operations or extracting one member from a multiple-valued field, can be performed only on the fields of Script nodes. The design of OO-VRML has concentrated on the encapsulation feature.

In general, an object-oriented programming language adopts one of two different ways of defining and creating objects. The one is class-based and the other is delegation-based. In class-based languages, such as C++, the implementation of an object is specified by its class and an object is created by instancing[2] its class. In delegation-based languages, objects are defined directly from other objects. DIVE [1, 6] system employs this approach. Since VRML is essentially a 3D scene-description language, it does not need to choose only one way as a general programming language does. OO-VRML is designed to take full advantage of both ways.

3 DEFINING A NEW NODE TYPE

This section covers the method of defining a prototype in OO-VRML, or prototyping, to introduce a new node type which corresponds to a class in object-oriented systems. Prototyping provides a class-based way of defining and creating objects. In this section, most of the object-oriented features incorporated into OO-VRML are explored.

3.1 Main Differences Between VRML And OO-VRML

In VRML, when a node receives an event, it merely assigns the value of the event to its associated field and generates new events if necessary. Thus, in order to add a general operation to a node, an independent Script node is inevitable for use. An Interpolator node may be put to use. However, it supports only linear keyframed animation for a specific field type.

The most apparent difference between VRML and OO-VRML is that OO-VRML no longer provides Script nodes and routes. Instead, event handlers of a node enable the node to take general actions. Event handlers are included in the prototype definition of the node and are executed when events are received. In most cases, VRML employs routes to hand over an event from one node to other nodes. In contrast, a node of OO-VRML sends an event directly to other nodes, not via routes, which conforms to the generally accepted principle in most object-oriented systems. VRML provides a way that events may be sent directly from one node to another. However, it must be a Script node with its `directOutput` field set to `TRUE` that sends such events. Therefore, there is no way of sending an event directly from a node, except the Script node, to other nodes. In OO-VRML, any node can send events directly to other nodes, not via routes.

For every language element of VRML, such as primitive node types, field types, or the `DEF` keyword, there exists a corresponding element in OO-VRML. Furthermore, one may assume freely that every element of OO-VRML not mentioned otherwise in this paper has exactly the same meaning as its corresponding element of VRML. For example, every field type of OO-VRML has the same meaning and the same syntax as the corresponding field type of VRML. Fields are considered private and cannot be changed by other nodes, while exposedFields are public and may be modified by other nodes. The `addChildren` field of the Transform node of OO-VRML has the same meaning as that of the Transform node of VRML.

3.2 Prototypes

Prototyping is a mechanism by which the set of available node types can be extended from an OO-VRML file. A prototype definition consists of the following.

- the `PROTO` keyword
- the name of the new node type
- the optional `extendible` keyword
- the prototype declaration which contains the following:
  - a list of eventIns and eventOuts
  - a list of exposedFields and fields with default values
- the prototype definition which contains the following:
  - zero or more prototypes
  - zero or more event handlers
  - zero or more nodes

An informal OO-VRML syntax for prototyping is shown below.

```
PROTO protoTypeName <extendible> [
  eventIn eventTypeName name
  eventOut eventTypeName name
  exposedField fieldTypeName name defaultValue
  field fieldTypeName name defaultValue
  ... ] {
    zero or more nodes
    zero or more prototypes
    zero or more event handlers
  }
```

An eventIn of OO-VRML has a different meaning from that of VRML. In VRML, an eventIn is associated with a field. In OO-VRML, an eventIn has its own event handler. If an eventIn `X` is declared in a prototype declaration, there must be an event handler with the same name in the prototype definition, as follows.

```
eventIn eventInType X # in the prototype declaration
HANDLER X {           # in the prototype definition
  ...
}
```

When a node receives an event for `X`, the event handler `X` is invoked. If the handler is not present in the prototype definition, it causes an error during parsing. An event handler of an OO-VRML node can carry out such works as the Script node of VRML performs. It describes an action that the node takes when a relevant event is received. Thus, an operation may be performed on a node only through an eventIn declared in its prototype, which conforms to the design criteria of object-oriented systems.

EventOuts have completely different semantics from those of VRML. An eventOut `Y` of a node may contain zero or more eventIns of other nodes.[3] If the eventOut is of field type T, all those eventIns are expected to be of type T. Assigning a value of type T to eventOut `Y` generates an event and delivers the event to all the eventIns that are contained in `Y`. Hence, an event is sent not via routes but directly to other nodes. The contents of an eventOut of a node, zero or more eventIns of other nodes, are given when instantiating the prototype. For example, suppose that an eventOut `Z` of type T of node A is given the following list of eventIns.[4]

```
Z [B.P C.Q D.R]
```

When assigning a value `V` to `Z`, three events are generated, one for each of the eventIns in `Z`. It is depicted in Figure 1.

Every eventOut of the primitive nodes of OO-VRML is activated as it is assigned a value on the same condition as in VRML. For example, the TouchSensor node of VRML generates events as the pointing device passes through any geometry nodes that are descendants of the TouchSensor node's parent group. The TouchSensor node of OO-VRML behaves in the same manner.

Fields are the parameters that distinguish a node from other nodes of the same type. The semantics of fields is the same as with VRML. An exposedField declares implicitly an eventIn with the same name, hence, also defines an event handler with the same name in the prototype definition.[5] Note that unlike VRML a corresponding eventOut is not declared. When the eventIn receives

[2] The notion of instancing in VRML is different to that in general object-oriented systems. In VRML, instancing means using the same instance of a node multiple times, which is called 'sharing' in other systems.

[3] The default value of an eventOut contains no eventIn.

[4] The syntax to access fields, eventIns, and eventOuts of a node is the same as with VRML.

[5] However, you cannot declare a field and an eventIn with the same name together in a prototype. This violates the name scoping rule of VRML, hence, also the name scoping rule of OO-VRML.

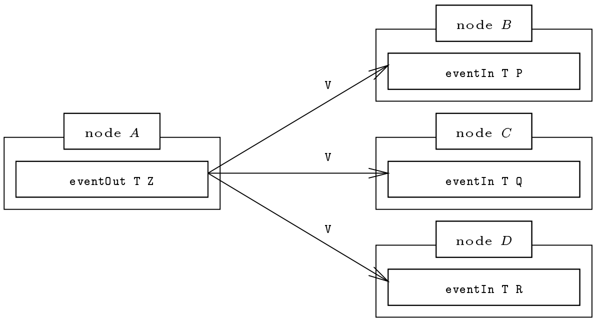

Figure 1: Multiple events sent through a common eventOut.

an event with value V, the value of the exposedField is changed to V. The event handler defined implicitly is assumed to perform this operation.

The optional extendible keyword plays an important role when the prototype is instantiated. It specifies whether we may attach additional event handlers to the new node or not. It also specifies whether the event handler defined in the prototype may be redefined or not. A more detailed explanation on the extendible keyword is presented in Section 5.

The first node appearing in the prototype definition is used to determine the type of nodes of this prototype, as with VRML. If there is no node in the prototype definition, the type of nodes of this prototype does not exist. Usually, such a prototype contains only event handlers in its prototype definition and is used to create nodes which can be regarded as the counterpart of the Script node of VRML. Any prototype found in the prototype definition holds good only inside the definition part, i.e. it is invisible from the outside of the prototype. The prototype name must be unique. It cannot rename a built-in node type or an already defined prototype.

3.3 Event Handlers

The event handler is used to program general operations of nodes. When a node receives an event through an eventIn, the event handler with the same name as the eventIn is invoked. Event handlers are built on the following syntax.

```
HANDLER eventHandlerName {
  field MFstring url []
  # any number of
  field fieldTypeName name defaultValue
}
```

Each event handler has its own programming language code, referenced by the url field, as with the Script node of VRML. The event handler's function is described in the url field. The url field may have a piece of inline language code or specify a URL (Uniform Resource Locator) which refers to a file. The way to fill the url field is explained in Section 4.

An event handler may have its own local fields that are visible only within the handler. These local fields may be used to store temporary computational results, values sent to eventOuts, or other information for a general purpose. Like fields in the prototype declaration, all the local fields of an event handler are persistent across event handler invocations. Note that exposedFields are not permitted in event handlers. The local fields may have their default values. They are initialized when the prototype is instantiated.

Every prototype in OO-VRML may define an event handler named initialize. The handler is called after the prototype is instantiated and before events are received. Likewise, an event handler named shutdown may be defined. The handler is called when a relevant node is deleted. Also an event handler named eventsProcessed may be defined, which is called after one or more events are processed. An associated eventIn for any of the above three event handlers is not required in the prototype declaration. The handlers initialize and shutdown correspond to the constructor and the destructor in an object-oriented language, respectively. The counterparts of these event handlers are found in the Script node of VRML.

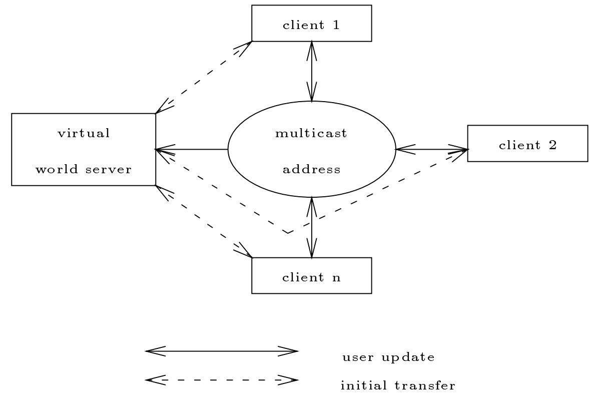

Figure 2: Multi-user system based on the multicast mechanism.

4 MULTI-USER ENVIRONMENTS

One of the basic design criteria of OO-VRML is that it should be suitable for the easy implementation of multi-user systems. The mechanism of event transmission among nodes in OO-VRML is much simpler and far more restricted than in VRML. Additionally, OO-VRML considers the contents of messages that should be distributed among clients participating in a shared virtual world when an event transmission between nodes occurs in a specific client. For this purpose, particular attention is paid to the url field of the event handler. In this section, we present the proposed structure of messages and describe fully the url field of the event handler.

4.1 Fundamental Connection Structures

To begin with, we consider fundamental connection structures for a virtual world server and its clients. At a rough approximation, they can be categorized into structures that are built on direct server-client connection [4, 9], and structures that are built on multicast mechanism [4, 5]. In the first approach, a client receives from the server all necessary update information about the world that arises from other clients. Since many clients communicate directly with the server, the server takes relatively heavy tasks and each client is given a relatively simple work. The central server can become a bottleneck of the multi-user system very quickly.

The second approach employs the multicast mechanism. In this approach, a client sends its messages directly to other clients through a common multicast group, not via the virtual world server. The server keeps the up-to-date record of the virtual world state and sends the current record to a new client trying to participate in the virtual world. This approach has already proven to be suitable for large scale multi-user virtual world systems, such as DIVE. However, the multicast mechanism is less reliable because any message sent by one client is not guaranteed to arrive at other clients within a specific time limit. Furthermore, the message may be lost during transmission.

In designing OO-VRML, the multicast mechanism is adopted as the basic model of the connection structure, although the connection structure has no direct relation with any OO-VRML feature. OO-VRML is designed to minimize the problem incurred by employing the multicast mechanism, such as lost messages and simultaneous arrival of more than one message for a common eventIn.

4.2 Message Transmission Among Clients

For the purpose of a clear discussion, we use the term 'message' to describe data transmitted from one client to another client or to the virtual world server. Message transmission is completely different from event transmission which refers to sending events from one node to other nodes within one client. A message has a piece of information on transmitting one or more new events generated in an event handler to other nodes within a specific client. Note that an event handler packs a message after the completion of its execution and sends it to other clients. A message originated in one client is delivered to all other clients to inform them

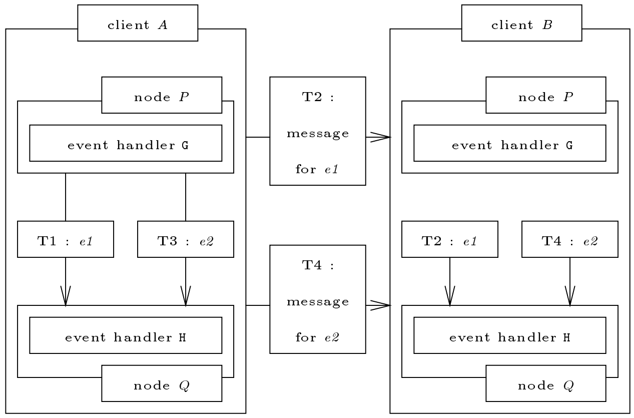

Figure 3: Transmission of two messages from client A to client B.

of a possible change in the shared virtual world. The recipient of the message invokes one or more event handlers which process the events contained in the message. These event handlers need not generate new events because messages for the new events will be delivered soon from the sender of the first message. One may wonder why a recipient of a message does not generate new events and process them for itself. However, for the purpose of consistency among clients, it should be permitted only to some special nodes, such as the TimeSensor node. Two messages are said to be of the same type if they are packed by one event handler.

Consider the following situation, which is presented in chronological order. The messages for events *e1* and *e2* are of the same type. It is depicted in Figure 3. In this case, the messages contain only one event.

T1 Event *e1* is transmitted from node P to node Q within client A.

T2 A message for event *e1* arrives at client B.

T3 Event *e2* is transmitted from node P to node Q within client A.

T4 A message for event *e2* arrives at client B.

In a normal case, the event handler `H` of client B processes event *e1* at T2 and event *e2* at T4, which results in a consistent state with client A (and probably with all other clients). However, serious inconsistency may occur among clients in the following case.

Case 1 The message for event *e1* is lost in transit.

It is probable that case 1 may occur because the multicast mechanism is not completely reliable. Since it is not guaranteed that processing event *e2* in client B puts right an inconsistent state of node Q due to the lost message for event *e1*, client B can not confirm whether or not its own record of node Q is consistent with others even after the message for event *e2* has been received. Therefore, it is a desirable requirement that processing all events in a message should put right a possibly inconsistent state resulting from previous missed messages of the same type. Consider another case for demonstrating the usefulness of the requirement.

Case 2 Both messages for events *e1* and *e2* arrive at client B at the same time.

Case 2 may be regarded as equivalent to a case where the message for event *e2* arrives at client B before the processing of event *e1* is completed. If the requirement described above is satisfied, we may discard event *e1* and process only event *e2*, which results in better performance of client B.

The requirement could be met by devising a sophisticated connection protocol for clients and the server, independently of the language on which the multi-user virtual environment system is based. However, if the language presents a specific feature supporting the requirement, it is much easier to implement such multi-user systems. Both the specification on the `url` field of event handlers of OO-VRML and the proposed structure of messages comply with this design principle. In the next section, we consider the contents of a message distributed among clients.

4.3 Contents Of A Message

A message should have all necessary information to process correctly a set of events generated in an event handler. Thus, a message has the following basic information for each event.

- event destination
  indicates to which eventIn of which node this event is directed.
- event value
- time stamp

Besides the basic information, a message should carry additional information about fields and exposedFields that are relevant to each event. Consider the following example of prototyping. The `url` field of event handler `X` is written in pseudo code.

```
PROTO [
  field SFVec3f A
  eventIn SFVec3f X ] {
    HANDLER X {
      url "
        # Let P be the parameter of event handler X
        # A[i] = A[i] + P[i], for i = 0,1,2
      "
    }
    ...
  }
```

In this example, the result of executing event handler `X` depends on the current value of field `A` as well as the value of parameter `P`.[6] Hence, any message containing an event sent to eventIn `X` must carry the value which field `A` of the sender has before the execution of event handler `X`. The recipient of the message must set field `A` to this value before executing event handler `X`. The reason for putting this additional information in the message is to guarantee that processing an event compensates for all events sent to the same eventIn that have been missed or not processed. Therefore, an OO-VRML parser should perform compile-time static analysis of each event handler and the server of a virtual world should make the outcome of the analysis available to all clients participating in the virtual world. A simple strategy is that a field is marked as static for an event handler if the event handler reads the field before writing a value to it. More efficient algorithms may be obtained by general data-flow analysis of event handlers.

Once this additional information is added to the message, it seems to follow naturally that the requirement described in the previous section is satisfied completely. In the next section, we present a case in which it is not true.

4.4 url Field Of The Event Handler

As mentioned before, an event handler's `url` field may specify a URL or have inline language code. An example of the first type of `url` is as follows. As with the `url` field of the Script node of VRML, the `url` field may contain multiple URLs.

```
  url "http://foo.com/sample.js"
```

We show the structure of the actual content of the `url` field with inline language code written in JavaScript. The general structure of the `url` field is shown below.

```
url "javascript:
  function main(parameter,time_stamp) {
    ...
  }
  function generate() {   # no parameter
    ...
  }
```

[6] We call field `A` static for event handler `X`.

```
  # additional functions invoked during the
  # execution of main()
"
```

When an event handler is called, function `main()` enters into execution first. Its first argument is the event value handed on to the event handler. The second is the time stamp. Function `generate()` is called after the execution of `main()` is completed. New events which are sent to other nodes may be generated during the execution of `generate()`. One may write in `main()` some statements that generate new events but all these events are entirely ignored. In other words, `generate()` is the only function where new events may be generated. Therefore, local fields are sometimes indispensable to store some computational results that are used as the values of new events.

This strict restriction is imposed principally in order to support the easy and efficient implementation of multi-user virtual environment systems by allowing those systems to catch new events easily and to handle them in a dedicated manner. Furthermore, a conditional event generation, such as the following pseudo code, should be avoided in `generate()`.[7]

```
  if (condition A) then generate event X;
  else generate event Y;
```

To see why, consider an example in which a conditional event generation is allowed.

T1 node $P \rightarrow$(event *X1*)$\rightarrow$node $Q \rightarrow$(event *Y1*)$\rightarrow$node R

T2 node $P \rightarrow$(event *X2*)$\rightarrow$node $Q \rightarrow$(event *Z1*)$\rightarrow$node R

The above situation is assumed to occur in client A. Recall that processing an event compensates for all missed events sent to the same eventIn. Homogeneous events *X1* and *X2* share a common eventIn of node Q. Hence, both are processed in a single event handler of node Q. Heterogeneous events *Y1* and *Z1* are generated in this handler. On receipt of events *X1* and *X2*, new events *Y1* and *Z1* are sent to different eventIns of node R, respectively. All the information about this situation should be delivered in the form of messages to another client B. However, in case the message for event *Y1* is lost in transit, node R of client B remains inconsistent with node R of client A until a new message for an event of the same kind as *Y1* arrives at client B, which may cause a serious inconsistency problem. Even if the message for event *Z1* arrives at client B, the inconsistency problem cannot be resolved because events *Z1* and *Y1* do not share a common eventIn of node R.

Therefore, for the reliability of the implementation, any conditional event generation should be avoided. It does not mean that a conditional event generation in `generate()` causes an error. Rather, it says that if there is no conditional event generation, it is guaranteed that the processing of a message puts right a possibly inconsistent state resulting from some missed messages of the same type. The creator of a virtual world should keep this point in mind when using OO-VRML.

Fields and exposedFields of a node are available to all event handlers in it by using their names. They may be read or written to. The local fields defined in an event handler are available only to the event handler. Their values are persistent across event handler calls.

Unlike VRML, an eventOut defined in a node can be only written to. Writing a value to an eventOut generates zero or more new events, depending on the number of eventIns contained in the eventOut. In OO-VRML, reading an eventOut returns neither the last value sent nor a default value, while in VRML reading an eventOut returns the last value sent. An OO-VRML parser should warn explicitly that it returns an undefined value on such a case. So, in order to retrieve the last value written to an eventOut, it should be kept in a specific field or exposedField. The event handler can assign a value to any eventIn or exposedField of a node to which it has a pointer. As with eventOuts, however, they cannot be read.

During the execution of `generate()`, assigning to an eventOut multiple times generates the same number of events in sequential order. In most cases, when packing a message, we only have to pay attention to the last value assigned to the eventOut because it is assumed that processing an event compensates for all events for the same eventIn that have been lost or not processed before. However, the preceding values cannot be ignored because they may affect some nodes.

[7] If events X and Y are sent to a common eventIn, it is not a conditional event generation.

5 OTHER CONSIDERATIONS

In this section, we cover other differences between OO-VRML and VRML that are not explained yet. An example is given to clarify the design principle of OO-VRML.

5.1 Representing A New Node

Representing a new node can be done easily by filling each field with an initial value as follows.

```
nodetype {
  fieldName initialValue
  exposedFieldName initialValue
  eventOutName initialValue
}
```

The default value of a field is used if an initial value is not specified. All event handlers in the prototype definition are automatically inherited to prototype instances.

In some case, it is desirable that a new event handler may be added to a node or an already defined event handler may be replaced by a new one. In OO-VRML, we do not need to define and create a new prototype only for this case. Instead, we may attach a new event handler to a node if its prototype is extendible. For extendible prototypes, an established event handler, including `initialize()`, `shutdown()`, and `eventsProcessed()`, may be overridden by a new event hander with the same name. More than one event handler with the same name may be attached but only the last handler is valid.

```
PROTO T extendible [...] {...}
T {
  ...
  eventIn SFBool test
  HANDLER test {
  }
}
```

To attach a new handler, the corresponding eventIn with the same name as the handler should be declared first. The eventIn specifies the type of the event value. This feature of OO-VRML is partially conformable to the basic design principle of delegation-based object-oriented languages. Note that new fields or exposedFields cannot be added to any prototype instance, whether the prototype is extendible or not. Group nodes are the only extendible basic node types of OO-VRML.

5.2 Node Name Scoping Rule

The `DEF` keyword defines a node's name and creates a node of that type. The `USE` keyword indicates that a reference to a previously named node should be used. In VRML, if multiple nodes are given the same name, then the last `DEF` encountered during parsing is used for `USE` definitions. In OO-VRML, the rule does not hold, namely, a node's name declared by using the `DEF` keyword becomes invalid outside the most enclosing block where the name is declared. For instance, the second `USE X` in the following example causes an error during parsing because the name `X` is no longer valid after the Transform node.

```
Transform {
  children [
    DEF X Transform {...}
    USE X   # OK
  ]         # Name X becomes invalid.
}
USE X       # Error
```

This scoping rule is adopted to support the encapsulation feature of object-oriented languages.

5.3 Sensor Nodes And Interpolator Nodes

Each event generated in Sensor nodes and Interpolator nodes is usually directed to other nodes so that it may have useful effects on the virtual world.[8] Since OO-VRML no longer provides routes, the destination of the event should be specified when representing those nodes. In the following example, the node T sends an event to nodes X and Y when necessary.

```
DEF X ...
DEF Y ...
DEF T TouchSensor {
  enabled FALSE
  hitNormal [X.inEvent01 Y.inEvent02]
}
```

5.4 An Example

The following prototype is used to define nodes which determine whether a given color contains a lot of red component. An example of the Script node of VRML defined for the same purpose is found in [3]. One could see the main differences of the two languages by examining this example.

```
PROTO T extendible [
  eventIn SFColor colorIn
  field SFColor currentColor 0 0 0
  eventOut SFBool resultEvent] {  # destination
    HANDLER colorIn {
      field SFBool result
      url "javascript:
        function main(c,ts) {
          # this function is called first when
          # an event colorIn is received.
          currentColor = c;
          if (currentColor[0] >= 0.5)
            result = true;
          else
            result = false;
        }
        function generate() {
          # A new event is generated.
          resultEvent = result;
        }
      "
    }
  }
DEF X ...
DEF Y ...
DEF myDetector1 T {
  resultEvent X.inEventName
}
DEF myDetector2 T {
  resultEvent [X.inEventName Y.inEventName]
}
```

6 DISCUSSION

Our object-oriented strategy for OO-VRML has an improvement on the current VRML architecture for the following reasons. First, the behavior of a node is completely described in its prototype definition or additional event handlers attached to the node when representing the node. Hence, one can build or reconstruct a virtual world in an object-oriented style. Interpreting a virtual world file is also easier in OO-VRML than in VRML because one has only to concentrate mainly on each node itself, not the relations between nodes. Second, OO-VRML presents a simplified and restrictive structure for the url field of the event handler as well as the basic structure of messages distributed among clients. The structures support partially the multi-user virtual environment systems based on the multicast mechanism. It facilitates the methodical implementation of reliable systems. Finally, compared with the VRML architecture, the OO-VRML architecture reduces the number of events passed among nodes to achieve general behavior of a node. It results from the elimination of the Script node and the route in OO-VRML. Sample cases for both VRML and OO-VRML are shown in Figure 4 and Figure 5, respectively. In Figure 4, event passing between nodes occurs twice before the event with value $f(\vec{x})$ reaches node B. In Figure 5, event passing occurs only once because the computation of $f(\vec{x})$ is accomplished inside node B. Therefore, the OO-VRML architecture reduces the network burden for a multi-user system because the number of messages sent by a client is decreased.

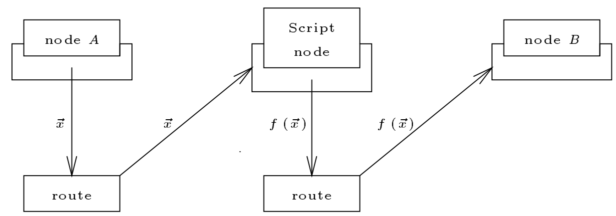

Figure 4: Passing events in the VRML architecture.

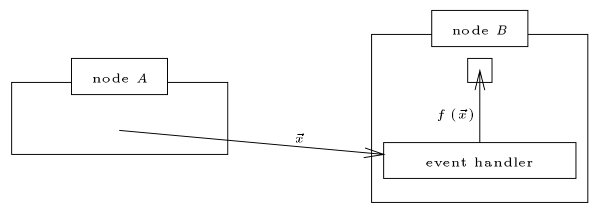

Figure 5: Passing an event in the OO-VRML architecture.

OO-VRML has a drawback that it has no language construct to prevent the conditional event generation. Additionally, the behavior of a node in OO-VRML may not be as general as in VRML because of the restrictive structure of the url field of the event handler. Employing a powerful script language could give a remedy for such problems.

7 CONCLUSIONS AND FUTURE WORK

We have designed an object-oriented 3D scene description language, OO-VRML, which is a derivative of VRML. The primary design criteria of OO-VRML are to enhance the object-oriented feature of VRML and to make it suitable for the easy implementation of multi-user systems. Event handlers of OO-VRML undertake the role of Script nodes and routes of VRML. The event handler is the vital ingredient of OO-VRML for its object-orientedness. The structure of the event handler's url field is designed to support multi-user environments. In OO-VRML, attaching general behavior to a node can be accomplished in an object-oriented way. Hence, it is easier than in VRML to build or interpret a complex virtual world file. Although the basic idea of OO-VRML is very simple, it is meaningful that VRML has evolved into OO-VRML which provides a new method for programming general behavior of nodes in an object-oriented style.

Our future work involves developing an OO-VRML-to-VRML translator which will be used to measure the utility of OO-VRML. The object-oriented facility of OO-VRML in creating a virtual world will be examined by testing a sufficient number of examples. The justification of the restriction forced upon the event handler will be performed along with the test process. An interesting research topic is about developing a methodology to handle general conditional event generations in the event handler. There may be many ways, from simple to complex, but the way which minimizes the network burden and resolves completely the problem of inconsistency among clients should be adopted. The proposed message transmission protocol should be further refined to release a formal specification on the protocol. The verification of the propriety of the protocol for multi-user environments will be carried out by implementing a simple system and testing the protocol on that system.

[8] Another sort of node with this property is the Collision node.

We believe that VRML will inevitably develop into a 3D scene description language supporting the multi-user environments and possibly object orientation. The next version of VRML may be far more advanced for this aspect than the current version. We hope that OO-VRML will be contributive to improving VRML in these directions.

References

[1] *DIVE 3.0.13 Files Specification.* "http://www.sics.se/dive/manual/dive_file_format.html".

[2] Andrea L. Ames, David R. Nadeau, and John L. Moreland. *The VRML Sourcebook.* John wiley & sons, Inc., 1996.

[3] Gavin Bell, Rikk Carey, and Chris Marrin. *The Virtual Reality Modeling Language Specification Version 2.0.* "http://vag.vrml.org/VRML2.0/FINAL/vrml2.ps.gz".

[4] Wolfgang Broll and David England. Bringing Worlds Together: Adding Multi-User Support To VRML. In *Symposium on the Virtual Reality Modeling Language*, pages 87–94, 1995.

[5] Donald P. Brutzman, Michael P. Macedonia, and Michael J. Zyda. Internetwork Infrastructure Requirements For Virtual Environments. In *Symposium on the Virtual Reality Modeling Language*, pages 95–104, 1995.

[6] Christer Carlsson and Olof Hagsand. DIVE - a Multi-User Virtual Reality System. In *Virtual Reality Annual International Symposium*, pages 394–400. IEEE, 1993.

[7] Kathleen Fisher and John C. Mitchell. *What Is An Object-Oriented Programming Language ?* "ftp://theory.stanford.edu/pub/kfisher/whatis-oop.ps".

[8] Kouichi Matsuda, Yasuaki Honda, and Rodger Lea. *Sony's Approach To Behavior And Scripting Aspects Of VRML: An Object-Oriented Perspective.* "http://www.csl.sony.co.jp/project/VS/proposal/behas
cri.html".

[9] Hwang Myung-gyu and Wohn Kwang-yeon. Multiparticipant 3D Browsing System On The WWW. In *Korea Human Computer Interaction Symposium*, pages 75–88, 1995.